Outils de génération pour le moteur de rendu GigaVoxels

Jérémy Jaussaud

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Outils de génération pour le moteur de rendu GigaVoxels

Jérémy JAUSSAUD

sous la direction de Fabrice NEYRET

À mon père. Merci.

Outils de génération pour le moteur de rendu GigaVoxels

Jérémy JAUSSAUD [*] [†] [‡]
sous la direction de Fabrice NEYRET [*] [†] [‡] [§]

Équipe MAVERICK

Août 2012 — 38 pages

[*] INRIA Rhône-Alpes
[†] Université Joseph Fourier
[‡] Laboratoire Jean Kuntzmann
[§] Centre national de la recherche scientifique

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Table des matières

1 Introduction

1.1 Contexte

Ce stage de fin d'études a été effectué au sein de l'équipe Maverick du Laboratoire Jean Kuntzmann, chez Inria, à Montbonnot. Cette équipe a développé la technologie « GigaVoxels » pour permettre l'exploration en temps-réel visuellement réaliste d'immenses volumes détaillés, eventuellement créés à la volée.

Cette technique vise un large type d'applications, depuis les jeux vidéos jusqu'aux effets spéciaux volumiques (avalanches, fumée, nuages), en passant par la visualisation enrichie d'objets astrophysiques (galaxie, nébuleuses, etc, ...) avec un projet financé par l'ANR en collaboration avec des partenaires industriels.

La phase de recherche exploratoire passée, nous avons maintenant besoin d'en tirer une plateforme (moteur de rendu) robuste, pour les chercheurs poursuivant les travaux sur le sujet et pour les collaborations industrielles ou académiques.

Il s'agit donc d'un travail en lien avec des utilisateurs-programmeurs experts, mais aussi avec plusieurs projets applicatifs (galaxie, paysage de nuages, scènes de jeux vidéos et d'effets spéciaux).

1.2 GigaVoxels

Sparse voxel octree

GigaVoxels[1] est un moteur de rendu écrit en CUDA [1] et développé initialement durant la thèse de Cyril Crassin[2]. Ce moteur se distingue principalement par le fait qu'il fasse un rendu à partir de voxels [2] au lieu de triangles comme le fait la quasi-totalité des moteurs existants.

Ces voxels sont stockés en cache par briques 3D de taille unique au sein d'un « *N-Tree* », le plus souvent un *octree*. Cette structure de données de GigaVoxels exploite le fait que les détails d'une scène se situent généralement au sein d'une interface entre du vide et un objet opaque (Fig. 1). Ainsi, bien qu'il s'agisse techniquement de rendu volumique, l'objectif n'est pas d'observer l'intérieur d'un objet comme cela peut être le cas en visualisation scientifique.

1. Par abus de langage, « CUDA » est également utilisé pour désigner les langages de programmation sus-cités et les API associées.
2. « *volume element* », par analogie avec « pixel » signifiant « *picture element* ».

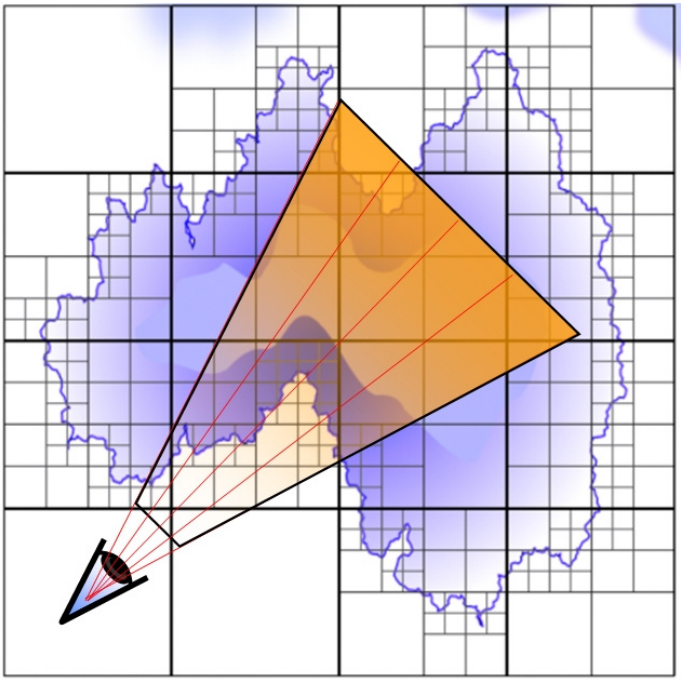


FIGURE 1 – Illustration du « *Sparse Voxel Octree* » de GigaVoxels, plus profond là où l'objet est plus détaillé.

GPU computing

Ce moteur est conçu pour pouvoir fonctionner entièrement sur GPU (Fig. 2), même s'il est possible de produire sur CPU les données à visualiser, à partir d'un fichier de données volumiques par exemple. Le cache est mis à jour automatiquement en fonction des besoins exprimés par les rayons.

Ceux-ci transmettent au « *cache manager* » l'usage qu'ils font de l'*octree* et, le cas échéant, les données qui leur manquent. Celles-ci sont alors produites et viennent remplacer dans le cache celles ayant été utilisées le moins récemment. Ainsi, on ne produit que les données nécessaires que l'on stocke dans le cache en *streaming*.

Le rendu

Le rendu se fait avec un « *voxel-based approximate cone tracing* ». Cela est très proche d'un lancé de rayons, mais on fait une approximation d'un cône lors de l'échantillonage des voxels (Fig. 3).

Un point crucial est le fait que les briques de voxels stockées dans l'*octree* sont de taille fixe, et que cette structure de données offre un *mipmap*[3] de la scène dont on

3. Le même objet à différentes résolutions, du plus grossier au plus fin.

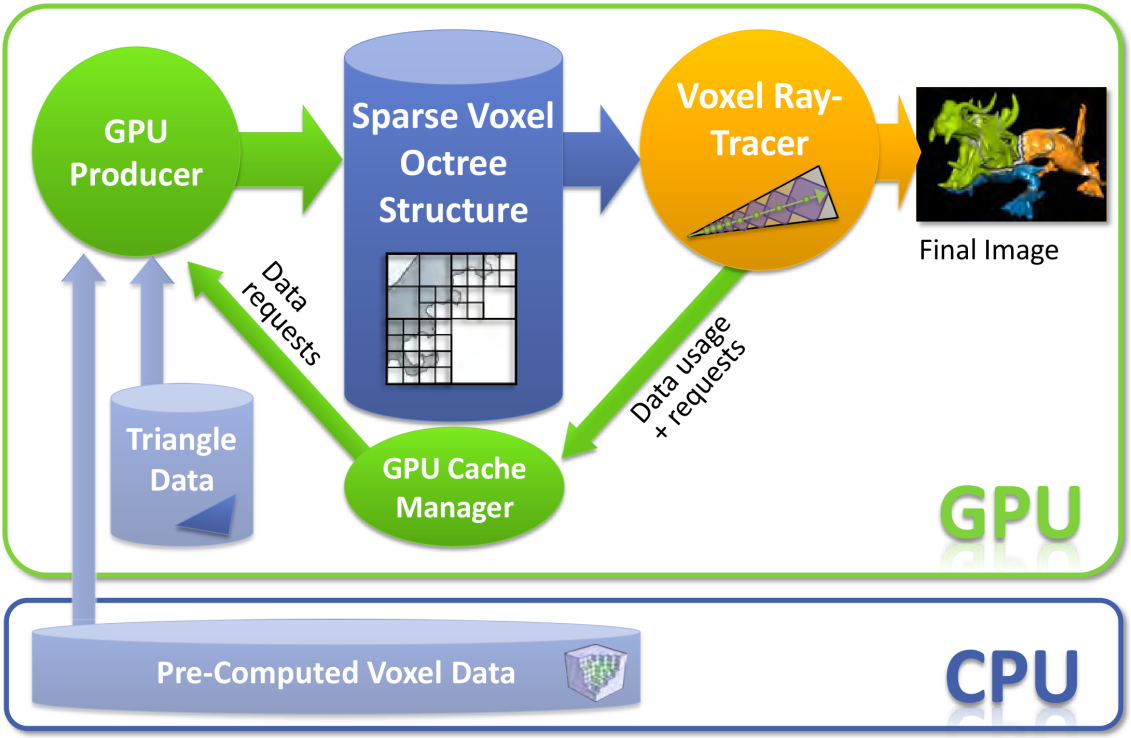


FIGURE 2 – Schéma du fonctionnement interne de GigaVoxels.

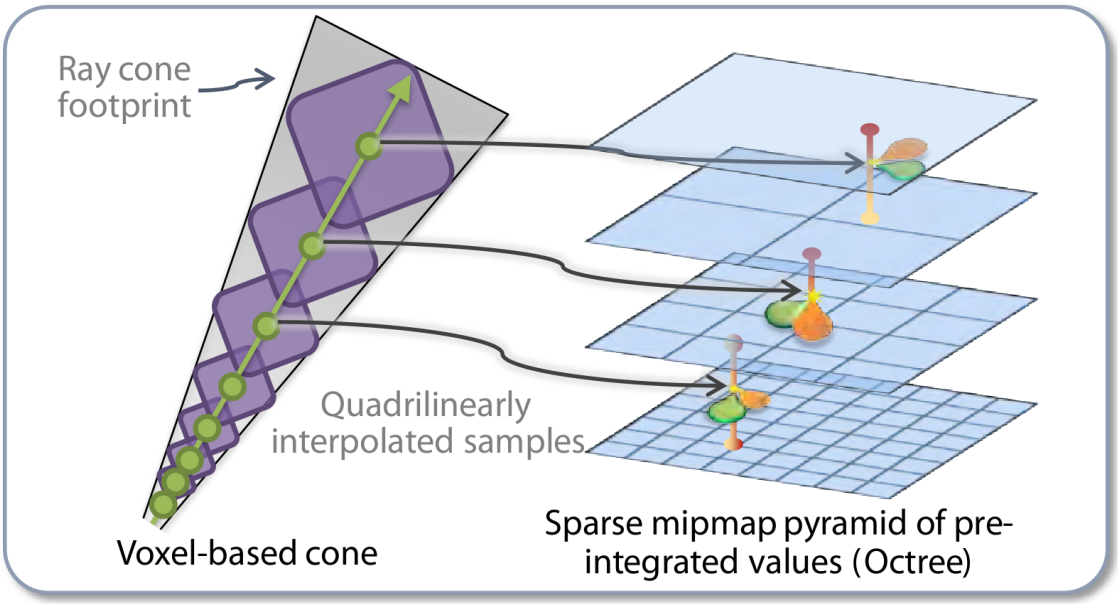


FIGURE 3 – Schéma représentant le « *voxel-based approximate cone tracing* » et de l'interpolation entre deux niveaux de résolution.

fait le rendu. On n'utilise pas nécessairement le niveau de profondeur maximal, mais seulement celui qui convient en fonction de la distance à la laquelle on se trouve de la caméra.

Le fait de ne pas utiliser nécessairement le niveau le plus résolu limite énormément la quantité de données qu'il est nécessaire de mettre dans le cache. De plus, les données sont interpolées trilinéairement au sein de chaque brique de voxels [4] mais également linéairement entre deux niveaux de résolution (Fig. 3). Cela évite l'*aliasing* et permet d'assurer la cohérence lors du passage d'un niveau de résolution à un autre.

1.3 Objectifs

GigaVoxels est une belle démonstration technique, mais la volonté que d'autres personnes puissent se l'approprier se heurte à certaines de ses limitations. Il est ainsi impossible de mêler GigaVoxels à une scène en triangle, standard à l'heure actuelle. Le manque d'exemples, de tutoriels mais surtout le manque d'API gênent à la fois son apprentissage et son utilisation.

Il s'agissait dans un premier temps de permettre de mêler GigaVoxels à une scène générée séparement pour, notamment, intégrer GigaVoxels à une scène en triangles (et vice versa) (Section 2, page 7) puis de mettre en œuvre un système permettant de faire du *geometry instancing* (Section 3, page 10). Il était également important aussi de développer quelques exemples servant à la fois de démonstrations et de tutoriels avec, notamment de la génération de bruit et d'hyper-textures (Section 4, page 12).

Pour faciliter l'apprentissage de GigaVoxels, de développer également une API de haut niveau permettant à l'utilisateur de découvrir le moteur sans avoir à se heurter immédiatement aux détails les plus techniques du moteur (Section 5, page 19). Cela a été l'occasion de concevoir une API permettant d'utiliser différents types de stockages pour la couleur et la profondeur : depuis un simple bloc de mémoire linéaire jusqu'à une texture OpenGL (Section 6, page 23), cela permettra de répondre aux besoin de l'utilisateur que celui-ci cherche les meilleures performances, ou qu'il veuille intégrer GigaVoxels à un pipeline complexe déjà existant.

4. On utilise pour cela une texture CUDA.

2 Mélange avec une scène en triangles

Une fonctionnalité préalablement manquante au moteur GigaVoxels est la possibilité de mêler ce rendu fait à partir de voxels avec une scène rendue séparément à partir de triangles.

2.1 Interprétation et génération d'un *Z-Buffer*

Le premier problème, qui était déjà quasiment résolu, est l'interprétation ou la génération d'un Z-buffer. Il faut pour cela traduire la profondeur par le Z-buffer pour la traduire en *model coordinates* pour qu'elle puisse être utilisée par GigaVoxels (Fig. 4). Le problème est simplifié du fait que l'on se restreint à des projections perspectives [5]. En OpenGL, la matrice M associée à une telle projection s'écrit sous la forme ci-dessous [6].

Considérant une telle matrice de projection, on peut en déduire facilement la relation liant z_e, la profondeur d'un point au plan de la caméra en *eye coordinates*, à z_c, la valeur correspondante en *clip coordinates*, puis enfin à Z, la valeur correspondante dans le Z-buffer. En notant $C = \frac{f+n}{n-f}$ et $D = \frac{2fn}{n-f}$, on obtient en *normalized device coordinates* $z_c = \frac{z_e C + D}{-z_e}$ et $z_e = \frac{D}{z_c + C}$. Enfin, passer du Z-buffer à ces coordonnées se fait ainsi : $z_c = 2Z - 1$.

$$M = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{t+b}{t-b} & 0 \\ 0 & \frac{2n}{t-b} & \frac{r+l}{r-l} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$\text{Au final, on obtient } \quad Z = \frac{1}{2}\left(1 - C - \frac{D}{z_e}\right) \quad \text{et} \quad z_e = \frac{D}{2Z - 1 + C} \quad .$$

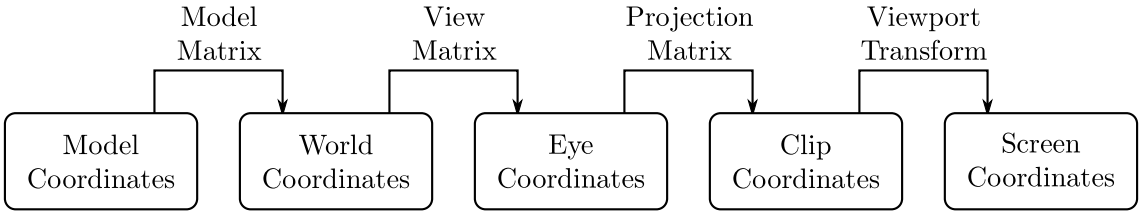


FIGURE 4 – Schéma représentant différents repères/espaces à manipuler dans un pipeline graphique.

5. Le *mipmap* de l'*octree* n'offre aucun avantage avec une projection orthographique.
6. `http://www.opengl.org/sdk/docs/man/xhtml/glFrustum.xml`

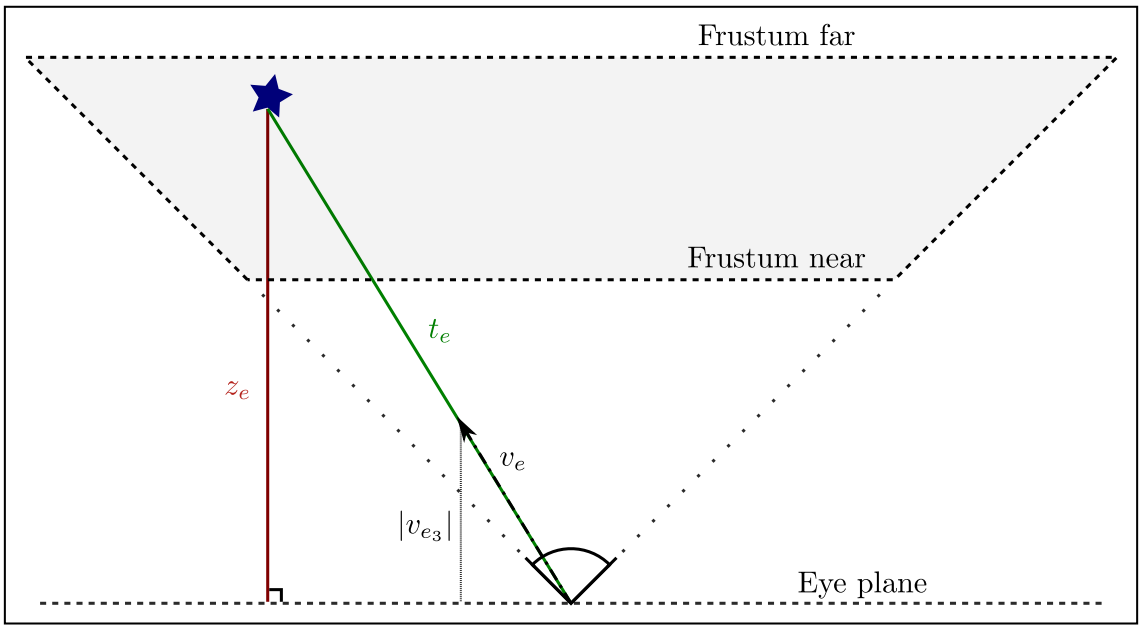


FIGURE 5 – Schéma représentant, en *eye coordinates*, z_e la distance au plan de vue (*eye-depth*) fournie par le Z-Buffer et t_e la distance à la caméra.

2.2 *Model/World/Eye Coordinates*

On sait obtenir z_e la distance au plan de vue en *Eye Coordinates* (*eye-depth*), mais GigaVoxels n'utilise pas directement cette information. À la place, il utilise t_m, la distance à la caméra en *model coordinates*.

On considère v_e et v_m ce vecteur directeur puis t_e et t_m la distance à la caméra, respectivement en *eye coordinates* et en *model coordinates*.

On a que $t_e = z_e \frac{\|v_e\|}{|v_{e_3}|}$ (Fig. 5), d'où $t_m = t_e \frac{\|v_m\|}{\|v_e\|} = z_e \frac{\|v_m\|}{|v_{e_3}|}$.

2.3 Redimensionnement isotropique

Il est ensuite intéressant de pouvoir redimensionner un objet GigaVoxels pour pouvoir l'intégrer correctement à une scène OpenGL, mais GigaVoxels doit impérativement utiliser un rayon directeur normalisé lors du rendu (Fig. 6). En effet, si celui-ci venait à être de norme inférieure à 1, on se retrouverait à faire des accès redondants à la texture stockant les voxels, ce qui serait coûteux. S'il venait à être supérieur à 1, on échantillonerait insuffisament cette même texture, introduisant de l'aliasing et brisant la cohérence spaciale des données. Il est donc nécessaire de continuer à normaliser ce vecteur.

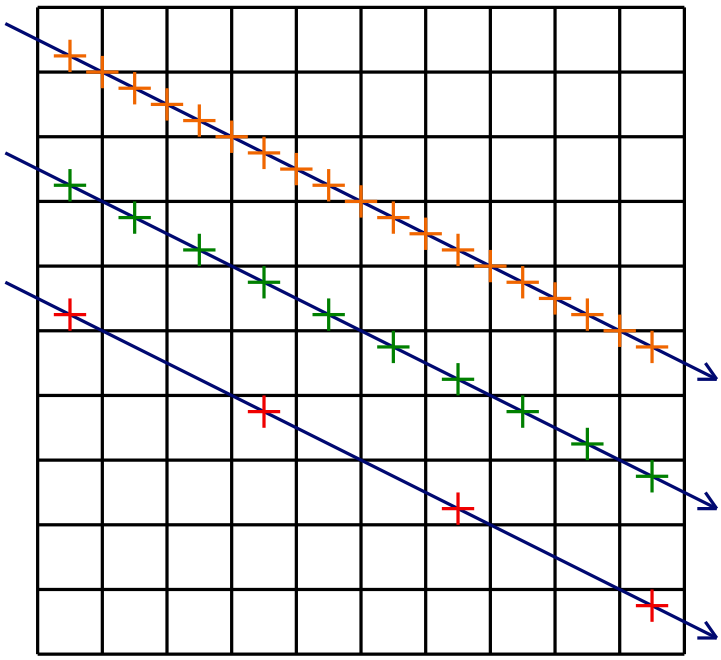


FIGURE 6 – Schéma représentant l'échantillonage de la texture en fonction de la norme du vecteur directeur v. Avec $\|v\| < 1$, on effectue des accès redondants. Avec $\|v\| = 1$, on echantillonne correctement. Avec $\|v\| > 1$ on n'échantillone pas suffisament la texture.

Le redimensionnement doit donc être pris en compte différemment. Le *shader* a besoin de connaître la grandeur du redimensionnement car, sans cette information, il est incapable de connaître les distances en *world coordinates* sans effectuer des calculs redondants et potentiellement coûteux.

C'est problématique car il en a non seulement besoin pour faire le *shading* d'objets translucides, mais également pour choisir correctement la précision de l'échantillonage. Sans connaître la valeur du redimensionnement, celui-ci ne peut que demander une résolution trop faible pour un objet agrandi par la matrice modèle, et une résolution trop importante pour un objet ayant été réduit.

Telle que les API du *shader* et du *kernel* étaient conçues, il était donc impossible de gérer efficacement une matrice modèle comprenant un redimensionnement puisque l'information devait être recalculée dans le *shader* pour chaque pixel de l'image. Ayant été décidé de se limiter à des redimensionnements isotropiques, il suffit de transmettre séparément le vecteur directeur et la grandeur du redimensionnement. Ainsi, le choix de résolution dans le *N-tree* et le *shading* peuvent tous deux être effectués correctement à peu de frais.

3 *Geometry Instancing*

Le *geometry instancing* consiste à offrir la possibilité d'instancier le même objet une multitude de fois en leur faisant subir différentes transformations. On peut par exemple se contenter de leur appliquer différentes matrices modèle (Fig. 7b).

Ce terme peut désigner différentes méthodes qui, même si elles permettent d'obtenir le même rendu, offrent des performances très différentes en termes de temps de calcul ou d'utilisation de l'espace mémoire.

On peut instancier différentes versions du même objet en mémoire pour ne faire qu'un seul et même rendu de ce tout. On peut également, au contraire, stocker l'objet qu'une seule fois pour en faire une multitude de rendu.

3.1 Un objet, plusieurs rendus

C'est la seconde méthode qui a été mise en œuvre. Cette solution ne permet pas d'obtenir un rendu aussi rapide que la première, mais elle bien plus économe en mémoire. Elle a également un autre avantage, lui aussi conséquent, c'est qu'elle permet de modifier certaines caractéristiques de l'objet bien plus facilement.

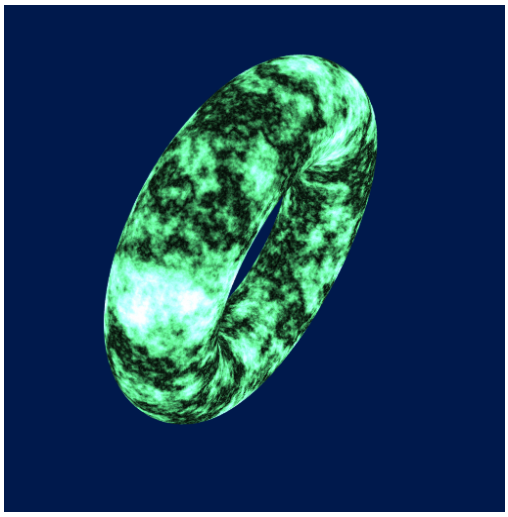
On peut en effet modifier la matrice modèle à la volée, ce qui permet de déplacer, translater et orienter les objets à volonté, même d'une image à la suivante. On peut par ailleurs modifier individuellement le *shading* de chaque objet. Ceci permet d'en modifier différentes caractériques comme le type d'ombrage utilisé, leur couleur de base, voire même leur texture.

3.2 Mise en œuvre

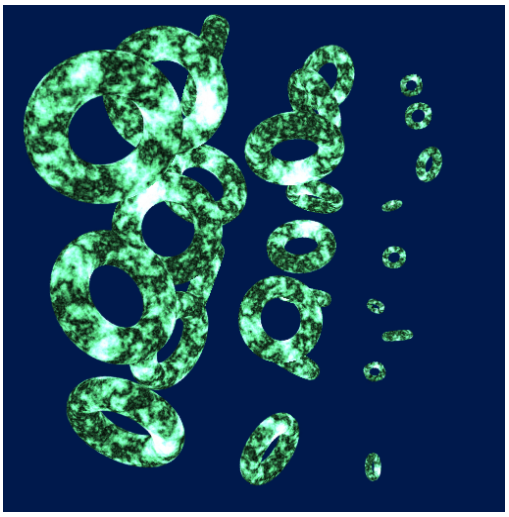
Il a été choisi d'itérer sur les différentes instances au sein du *kernel* (voir annexe A) effectuant le rendu, sans pré-traitement de la liste de matrices, ce qui constitue la méthode la plus rapide à mettre en place.

Le prototype qui a été développé nécessite non pas les matrices modèle, mais directement leurs inverses[7] qui sont utilisées au sein de GigaVoxels. On obtient de bonnes performances lorsque l'on a un nombre relativement restreint d'objets (Fig. 7), mais on est incapable de faire le rendu d'un très grand nombre d'objets comme cela serait nécessaire pour une forêt.

7. Les caractériques de la matrice modèle garantissent que son inverse existe.



(a) Rendu d'une seule instance d'un tore, 103 images par seconde.



(b) Rendu de 27 instances du même tore, 74 images par seconde.

FIGURE 7 – Rendu d'un tore puis de plusieurs à l'aide du *geometry instancing*.

3.3 Développements futurs

Le *renderer* qui a été développé est fonctionnel, mais il demeure un protoype manquant d'une API facilitant son usage. Son principe simple a permis de le développer assez rapidement pour obtenir de bonnes performances avec un nombre relativement restreint d'objets. Il fournit ainsi un temps de référence qui pourra être utilisé lors de la mise en place d'une méthode plus évoluée.

4 Bruit et hyper-textures

Lors du rendu d'un objet, on a souvent recours à différentes techniques pour ajouter des détails aux objets sans avoir à les stocker en mémoire. On peut par exemple raffiner un maillage (en apparence) en plaquant une texture 2D sur celui-ci, ou en le « taillant » directement dans une texture 3D.

Cette dernière solution est directement applicable dans le cas d'un rendu volumique comme celui qui est effectué dans GigaVoxels. Le bruit est calculé à la volée et peut être utilisé, à titre d'exemple, pour déterminer la couleur de l'objet (Fig. 8), ou pour perturber la normale.

GigaVoxels manquait jusque-là de fonctions permettant de générer ce genre de bruit, il a donc fallut les développer pour qu'elles puissent être utilisées facilement et efficacement à la fois sur CPU et sur GPU.

4.1 Somme de bruit fractale

Une somme de bruit fractale consiste à sommer une suite d'« octaves » elles-mêmes calculées à l'aide d'un bruit de base. Chacune de ces octaves possède une amplitude moindre et une fréquence supérieure à celles de la précédente, ces deux variables évoluant selon une suite géométrique.

Définition

On considère $noise : \mathbb{R}^3 \longrightarrow \mathcal{F}$ une application générant du bruit d'amplitude a. Soit f_n sa somme fractale sur $N \in \mathbb{N}$ octaves de persistence $p \in]-1, 1[$:

$$\begin{array}{rccc} f_N : & \mathbb{R}^3 & \longrightarrow & \mathcal{F} \\ & x & \longmapsto & \sum_{i=0}^{N-1} p^i \cdot noise(2^i \cdot x) \end{array}$$

Optimisation

Pour générer ce bruit efficacement, il a été proposé[3] une optimisation consistant à stopper la somme fractale lorsqu'il est certain que le reste de celle-ci ne changera pas la couleur finale (Fig. 9).

On suppose d'une part qu'en deça d'un intervalle $\mathcal{I}$ donné, et au-delà d'autre part, la couleur finale obtenue à partir de ce bruit sera identique. On borne donc le reste de la somme fractale que l'on compare avec cet intervalle $\mathcal{I}$.

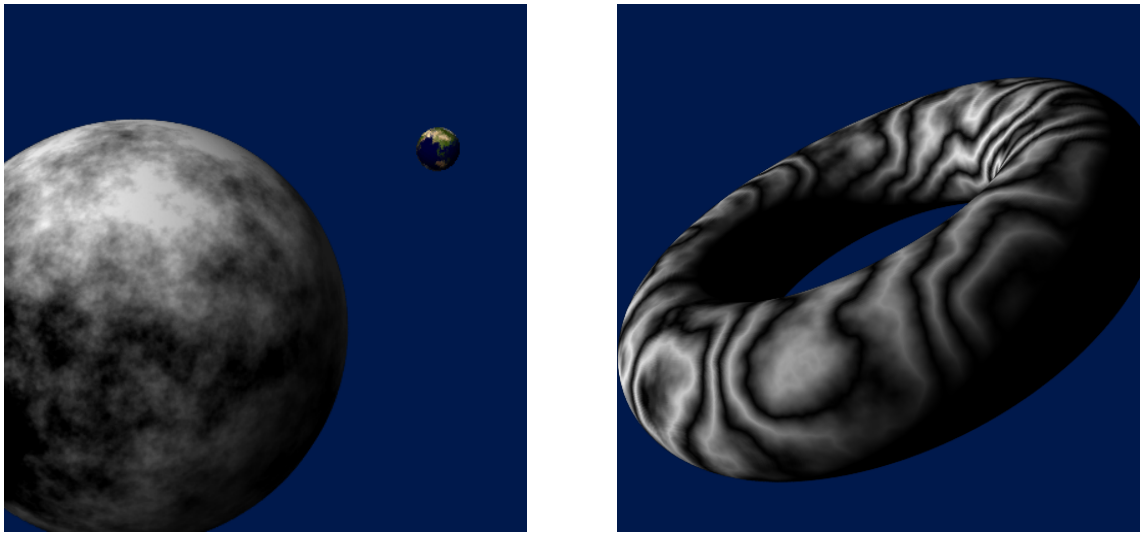


FIGURE 8 – Utilisation de textures 3D générées procéduralement à l'aide de différents bruits pour calculer la couleur d'une sphère puis d'un tore.

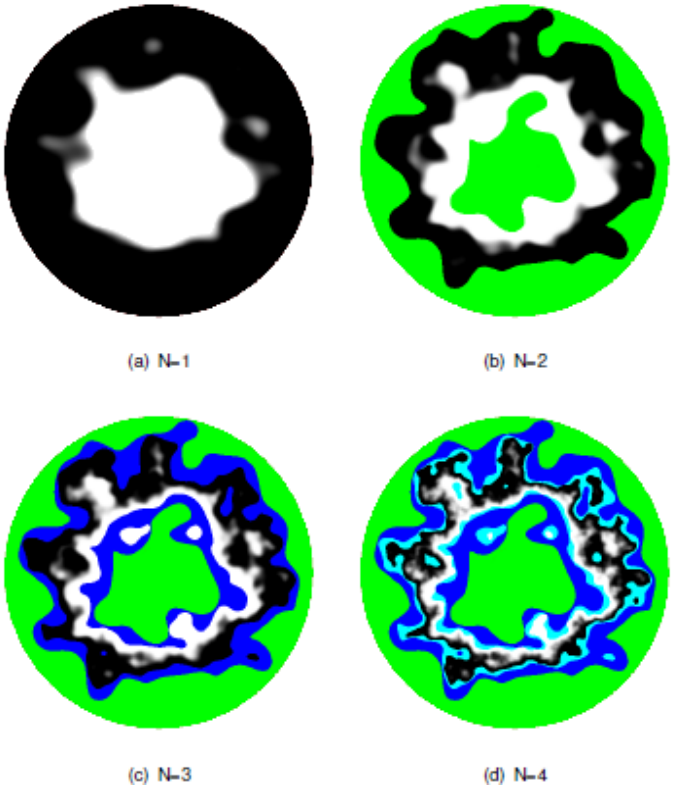


FIGURE 9 – Illustration de l'optimisation effectuée. Seule une octave a dû être calculée pour les pixels verts, deux pour les pixels bleus et trois pour les pixels cyans.

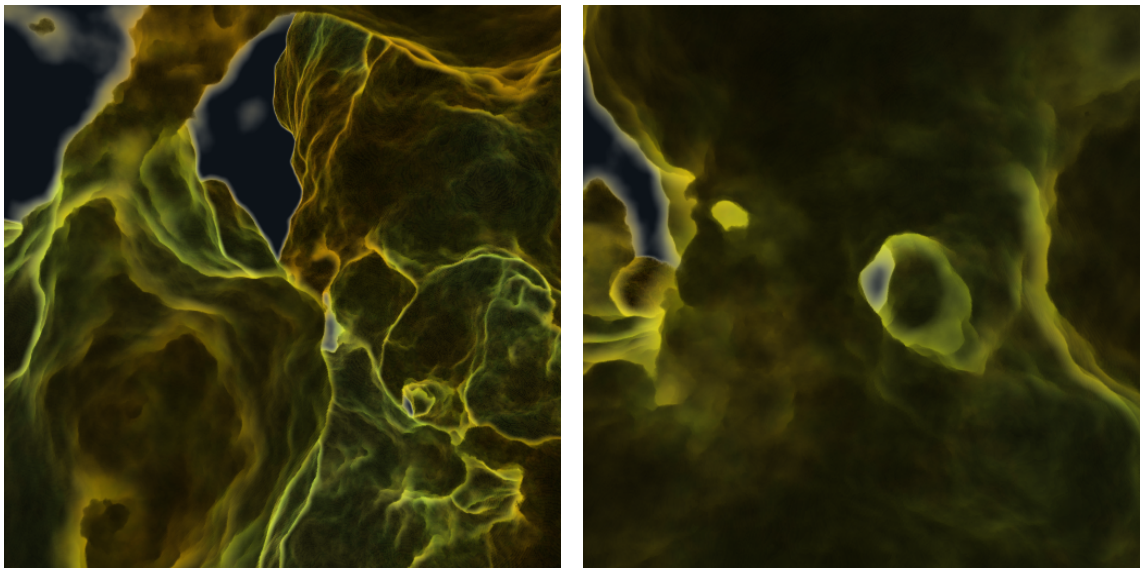


FIGURE 10 – Génération procédurale d'une hypertexture représentant une nébuleuse, avec une somme de bruit de Perlin fractale.

Cette optimisation était présentée[3] avec une persistence égale à 1/2 et un bruit de base d'amplitude 1, mais elle est facilement généralisable à une persistence $p \in]0,1[$ et une amplitude de base $a \in \mathbb{R}^+$.

$\forall x \in \mathbb{R}^3 \quad \forall n \in \mathbb{N}$ tel que $n < N$, on a que :

$$\begin{aligned} |f_N(x) - f_n(x)| &= \left|\sum_{i=0}^{N-1} p^i \cdot noise(2^i \cdot x) - \sum_{i=0}^{n-1} p^i \cdot noise(2^i \cdot x)\right| \\ &= \left|\sum_{i=n}^{N-1} p^i \cdot noise(2^i \cdot x)\right| \\ &\leq \sum_{i=n}^{N-1} |p^i| \cdot a \\ &\leq a \cdot \left(\frac{1-|p|^N}{1-|p|} - \sum_{i=0}^{n-1} |p|^i\right) = \mathcal{D} \end{aligned}$$

D'où $f_N(x) \in [\mathcal{B}_{min}, \mathcal{B}_{max}] = \big[f_n(x) - \mathcal{D}, \quad f_n(x) + \mathcal{D}\big]$.

Alors :

1. Si $\mathcal{I} \cap [\mathcal{B}_{min}, \mathcal{B}_{max}] = \emptyset$, on peut stopper la somme fractale puisque cela n'aura aucune influence sur le résultat final.
2. Si $[\mathcal{B}_{min}, \mathcal{B}_{max}] \subset \mathcal{I}$, il est plus avantageux de finir la somme fractale en ne faisant plus aucune vérification. Le gain supplémentaire est mesurable, même s'il est moins important que celui de la première optimisation.

Mise en œuvre

La mise en œuvre fonctionne à la fois sur CPU et GPU, et permet de renseigner jusqu'à quatre intervalles successifs. Ceux-ci sont passés en paramètre à l'aide de

deux *built-in vectors* (voir section 4.3, page 16), l'un pour les bornes inférieures et l'autre pour les bornes supérieures des intervalles.

Il peut être intéressant pour l'utilisateur de pouvoir générer un bruit fractal non fenêtré, au quel cas ces vérifications se mettent à coûter plus qu'elles ne rapportent. De la même manière, il peut être également intéressant de pouvoir désactiver la seconde vérification, d'autant plus que celle-ci est susceptible de briser des *warps* (voir annexe A).

L'optimisation permet au final, sur une grille régulière, de générer cinq fois plus vite le bruit utilisé pour générer la figure 10.

4.2 Hypertexture

Une hyper-texture est un cas particulier de texture 3D. Celle-ci est composée de trois types de région différentes : un cœur dense, du vide et, enfin, la couche de matière semi-transparente se trouvant entre les deux. Les détails se situent principalement au sein de cette interface entre le vide et la matière opaque (Fig. 10).

L'optimisation que l'on a mise en œuvre permet de gagner du temps lors du calcul de la somme fractale. Elle permet également de gagner une information supplémentaire sur le bruit qui vient d'être généré. En effet, si la somme a été interrompue par l'optimisation, alors on sait que le bruit généré se trouve soit au cœur de l'objet, soit dans le vide (Fig. 9).

Cette information est très importante pour GigaVoxels, et il est prévu de pouvoir facilement propager celle-ci au *cache* du moteur depuis le *producer* de données.

Grâce à cela, on peut marquer comme « terminales » les briques de voxels qui se situent au cœur de l'objet, ce qui évite d'avoir à calculer inutilement leurs subdivisions. On peut ensuite marquer comme « vides » les briques se trouvant en dehors de l'hypertexture (Fig. 11).

On exploite ainsi sans frais supplémentaire la structure d'*octree* de GigaVoxels pour économiser la mémoire dans le *cache* et le temps de calcul nécessaire au rendu.

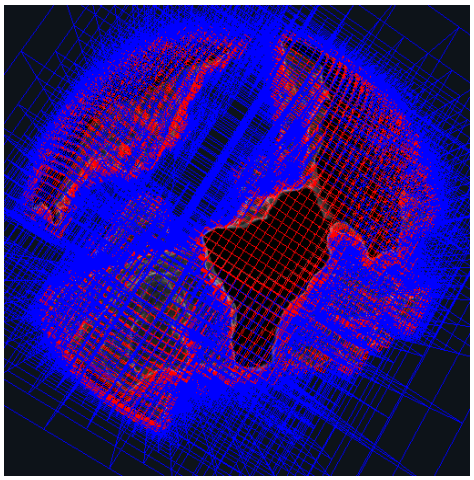


FIGURE 11 – Visualisation de l'Octree. Les briques normales sont en rouge, les vides sont en bleu.

4.3 *Built-in vectors*

Les *built-in vectors* utilisés pour l'optimisation de la somme fractale se manipulent comme de simples structures contenant de un à quatre éléments d'un type donné. Par exemple : `float4` contiendra quatre membres `x`, `y`, `z`, et `w` de type `float`.

Il s'agit bel et bien de types intrinsèques au langage et ils ont des avantages comparé à une structure classique. Certains jouissent en mémoire d'alignements inférieurs à ce que l'on obtiendrait en les redéfinissant à la main, et certaines opérations sur ceux-ci peuvent se traduire en une seule instruction PTX [8]. Il y donc bel et bien un intérêt à les utiliser.

Wrapper

Les vecteurs CUDA sont de simples structures sans aucune méthode, il n'est possible d'accéder à leurs membres que par leur nom, et pas par leurs indices. Cela a été l'occasion de développer `BuiltInVector`, une classe héritant de ces vecteurs pour leur adjoindre des méthodes et, notamment, l'opérateur *array subscript*. Ce dernier ne devrait être utilisé qu'avec un indice connu à la compilation, mais il permet dans ces conditions d'utiliser ces vecteurs comme on le ferait avec un tableau statique.

Ceci permet également de très facilement surcharger des méthodes, dont les opérateurs arithmétiques, et ce indépendamment de la taille des vecteurs (Fig. 12). On obtient de cette manière une alternative aux en-têtes du « *GPU Computing SDK* » [9]. Ces derniers faisaient ces surcharges, mais leur utilisation obligeait jusque-là l'utilisateur de GigaVoxels à installer plus de 300 Mo de fichiers qui n'étaient par ailleurs pas utilisés.

Mise en œuvre

Vous pouvez d'ores et déjà consulter le diagramme de classes (Fig. 13 page 18).

La classe `BuiltInVectorBase` fourni le type de *built-in vector* et la fonction constructeur associée, tous deux fournis par CUDA. Elle hérite également du vecteur CUDA qui devient la classe de base de la hierarchie, permettant à la classe de conserver l'alignement particulier de ce dernier.

La classe `BuiltInVector_Impl` met en œuvre l'opérateur *array subscript* et les opérateurs arithmétiques ; ses constructeurs sont privés et la classe `BuiltInVector` est déclarée `friend`.

8. Le langage assembleur exécuté sur le GPU.
9. `http://developer.nvidia.com/cuda/gpu-computing-sdk`

```
__host__ __device__ __forceinline__
BuiltInVector<T,N>
BuiltInVector_Impl<T,N>::operator+( const cudaType &v ) const
{
    BuiltInVector<T,N> result;
    BuiltInVector<T,N> w(v); // wrap the parameter
    #pragma unroll // force NVCC to unroll the loop
    for ( int j = 0 ; j < N ; j++ )
    {
        result[j] = (*this)[j] + w[j];
    }
    return result;
}
```

FIGURE 12 – Mise en œuvre d'un opérateur arithmétique. Cette définition est valable pour les 48 types différents de *Built-in vectors*, quelles que soient leurs tailles. La directive « `#pragma unroll` » permet de s'assurer que la boucle est bien déroulée[4][5].

Cette dernière dispose de différents constructeurs et des méthodes nécessitant d'être spécialisées selon la taille du vecteur, que ce soit à cause de leurs prototypes ou pour des questions de performance. C'est cette classe et seulement cette classe qui est destinée à être manipulée par l'utilisateur.

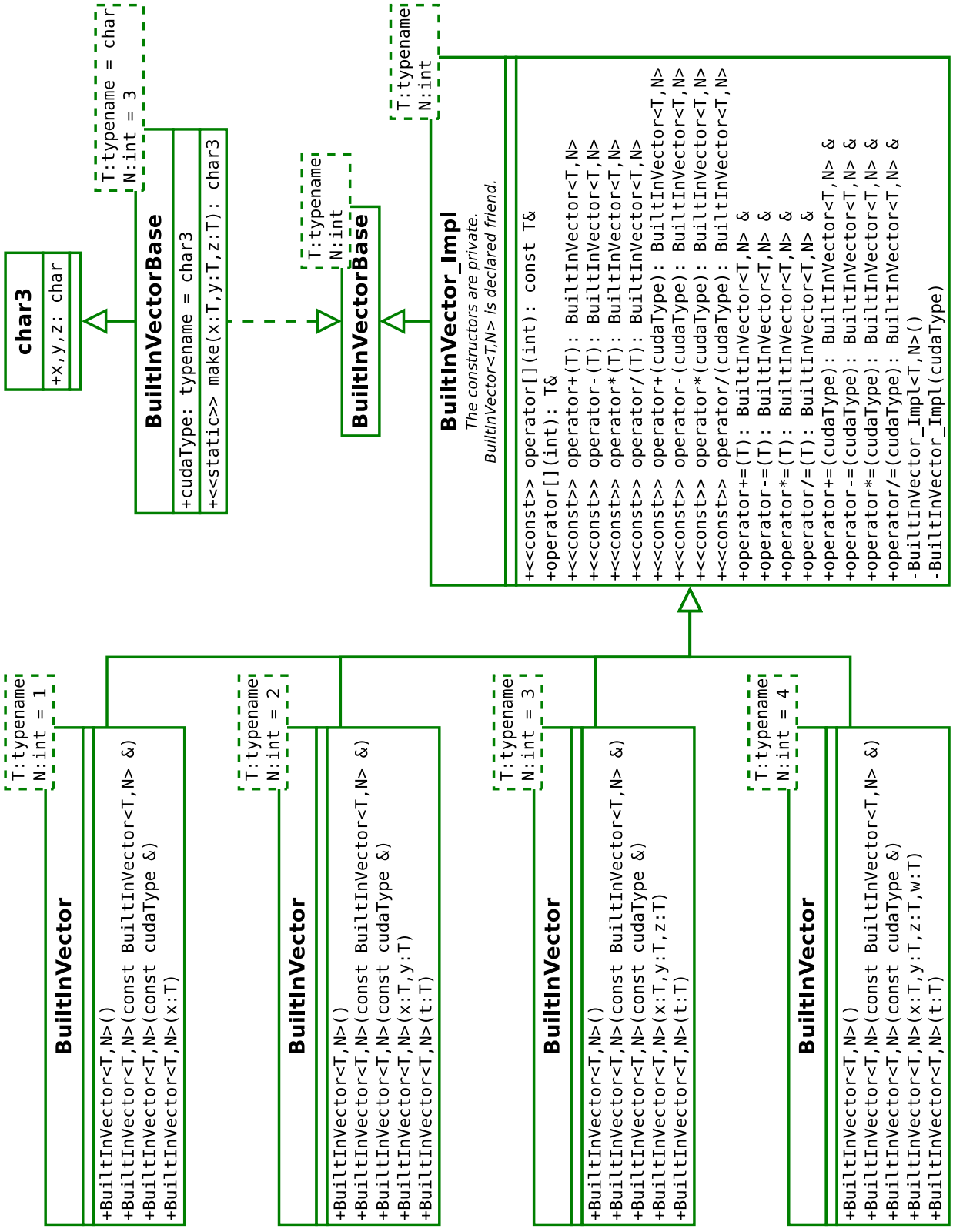


FIGURE 13 – Diagramme de classes de `BuiltInVector`.

5 API de haut niveau

Étant possible de mélanger GigaVoxels à une scène en triangles, il fallait développer une API de plus haut niveau permettant de mettre en place un exemple le plus facilement possible car, jusque là, c'étaient plus de 600 lignes de codes qui se trouvaient dupliquées dans chaque exemple existant.

Cela obligeait également l'utilisateur à manipuler directement CUDA et OpenGL, le confrontant à des détails de l'implémentation qui ne devraient pas le préoccuper. Il fallait pouvoir lui épargner cela, le temps qu'il puisse découvrir le moteur en apprenant peu à peu les différents concepts qui entrent en jeu.

5.1 Pipeline GigaVoxels

La classe `GvPipeline`

Jusqu'à présent, l'utilisateur devait lui-même créer les différents objets du pipeline GigaVoxels, les initialiser puis les détruire séparément à la fin du proparamme. Cela ouvrait la porte à des erreurs de programmation mais était surtout désagréable à utiliser. Il fallait donc développer une classe s'occupant de faire tout cela pour l'utilisateur. Celle-ci prend en paramètres *template* les types des éléments du Pipeline et s'occupe de les allouer et de les initialiser correctement.

On obtient déjà un gros gain d'ergonomie à peu de frais, mais cela ne suffit pas tout à fait. GigaVoxels fait un usage particulièrement intensif des *templates*, et certains paramètres *templates* du pipeline sont des classes que l'utilisateur doit lui-même définir [10]. Il est par conséquent impossible d'instancier le pipeline une fois pour toutes au sein de la bibliothèque GigaVoxels.

L'utilisateur doit donc non seulement déclarer manuellement les types des objets du pipeline, mais également les instancier, implicitement, ou explicitement dans le cas d'une compilation séparée.

Déclaration

Le problème a été résolu en créant une structure, sans attribut ni fonction, qui s'occupe de construire les différents types du pipeline de façon cohérente (Fig. 14). Ce code peut être assez rebutant pour qui n'est pas à l'aise avec les *templates*, mais son utilisation demeure bien plus simple que de devoir redéfinir chaque type séparément comme c'était le cas juqu'à présent.

10. Le « *producer* » et le « *shader* ».

```
template<
    template <typename,typename,typename,typename,typename,typename,typename,
        typename> class TPipelineType,
    typename TNodeRes, typename TBrickRes, typename TDataList,
    template <typename,typename,typename,uint> class TVolumeTreeType,
    template <typename,typename,typename,typename> class TVolumeTreeCacheType,
    template <typename,typename,uint,typename> class TProducerType,
    typename TSampleShader,
    template <typename,typename,typename,typename> class TVolumeTreeRendererType
>struct GvMakePipelineImpl
{
    /// border size
    enum { BrickBorderSize = 1 };

    /// The 3D Resolution of the nodes of the N-Tree
    typedef TNodeRes NodeRes;

    /// The 3D Resolution of the bricks of the N-Tree
    typedef TBrickRes BrickRes;

    /// The total size of a brick as stored (with a border) in the GigaVoxels cache
    typedef GvCore::StaticRes3D
    <   BrickRes::x + 2 * BrickBorderSize,
        BrickRes::y + 2 * BrickBorderSize,
        BrickRes::z + 2 * BrickBorderSize
    >   RealBrickRes;

    /// The complete type of the VolumeTree used in the pipeline
    typedef TVolumeTreeType
    <   TDataList, NodeRes, BrickRes, BrickBorderSize
    >   VolumeTreeType;

    /// The complete type of the Producer used in the pipeline
    typedef TProducerType
    <   NodeRes, BrickRes, BrickBorderSize, VolumeTreeType
    >   ProducerType;

    /// The complete type of the VolumeTreeCacheType used in the pipeline
    typedef TVolumeTreeCacheType
    <   VolumeTreeType, ProducerType, NodeRes, RealBrickRes
    >   VolumeTreeCacheType;

    /// The complete type of the Shader used in the pipeline
    typedef TSampleShader SampleShader;

    /// The complete type of the VolumeTreeRenderer used in the pipeline
    typedef TVolumeTreeRendererType
    <   VolumeTreeType, VolumeTreeCacheType, ProducerType, SampleShader
    >   VolumeTreeRendererType;

    /// The complete type of the Pipeline itself
    typedef TPipelineType
    <   NodeRes, RealBrickRes, TDataList, VolumeTreeType, VolumeTreeCacheType,
        ProducerType, SampleShader, VolumeTreeRendererType
    >   PipelineType;

    /// The complete type of the Pipeline itself
    typedef PipelineType Result;
};
```

FIGURE 14 – Mise en œuvre de la structure s'occupant de construire les différents types consituant un pipeline GigaVoxels.

Cette structure peut notamment être elle-même utilisée par une structure fille qui s'occupera d'utiliser des classes par défaut là où, de toute façon, GigaVoxels ne propose pas encore d'alternative. Au final, la déclaration du pipeline et des types sous-jacents peut se faire très simplement (Fig. 15a).

Instanciation

Pour séparer la compilation du pipeline GigaVoxels de la compilation du code utilisant ce dernier, il faut demander explicitement son instanciation. Il suffit pour cela, au sein d'une unité de compilation dont NVCC[11] aura la charge, de faire appel à une *macro* fournie dorénavant par GigaVoxels (Fig. 15b).

Celle-ci s'occupe de faire les instanciations nécessaires au bon déroulement de l'édition de liens. En pratique, elle doit instancier explicitement le pipeline lui-même, mais aussi chacune de ses sous-classes succeptibles d'être manipulées depuis l'extérieur de l'unité de compilation (le *renderer*, le *cache*, etc...). Cette macro est d'autant plus indispensable qu'il est impossible de faire ces instanciations explicites grâce aux *typedefs* construits précédemment. Il est en effet nécessaire de fournir le type élaboré que représente chacun d'entre eux.

La macro obtenue serait difficilement maintenable telle quelle. Toutefois, si l'on devait modifier l'API interne à GigaVoxels, alors il suffirait simplement de la reconstruire à partir de son alter ego `GvMakePipelineImpl`.

5.2 *Hello world !*

Tous les outils de base étant là, il a été rapide de développer un second pipeline permettant de facilement intégrer GigaVoxels à une scène OpenGL. Ce nouveau pipeline est déclaré et instancié tout aussi facilement que le précédent et il offre, en sus, une méthode publique `draw` prenant en paramètre la matrice modèle à utiliser (Fig. 15c), avec la matrice identitée par défaut.

Tout l'interfaçage avec OpenGL est fait par le pipeline, il copie le *buffer* de profondeur dans un PBO en entrée et écrit dans une texture en sortie. Cette dernière est enfin affichée à l'écran à l'aide d'un *quad* texturé[12].

Cette méthode offre de très bonnes performances et est, surtout, peu contraignante pour l'utilisateur. Ce dernier n'est par exemple pas obligé d'utiliser de *render buffer* s'il n'en a pas besoin et, dans le cas contraire, il peut le faire sans être gêné par le pipeline.

11. Le compitateur CUDA créé et maintenu par NVIDIA.
12. À noter qu'il faut veiller à désactiver l'éclairage lors de l'affichacge du quad texturé.

```
/// Defines the type list representing the content of one voxel
typedef Loki::TL::MakeTypelist< uchar4, half4 >::Result DataType;

/// Type of the GigaVoxels pipeline
typedef GvUtils::GvMakeGLSimplePipeline <
    2, 8, // resolution of a node tile, then a brick
    DataType, // type list representing the content of one voxel
    Producer, Shader // user defined producer and shader
>::Result
GigaVoxelsPipeline;
```

(a) Déclaration d'un pipeline GigaVoxels

```
GV_INSTANTIATE_GLSIMPLE_PIPELINE(2, 8, DataType, Producer, Shader)
```

(b) Instanciation explicite d'un pipeline GigaVoxels

```
void SampleViewer::init()
{
    if ( glewInit() != GLEW_OK )
        exit( 1 );

    // GigaVoxels pipeline initialization
    _GigaVoxels->init();

    // Viewer initialization
    setMouseTracking( true );
    setAnimationPeriod( 0 );
    startAnimation();
}

void SampleViewer::draw()
{
    glClearColor( 0.0f, 0.1f, 0.3f, 0.0f );
    glEnable( GL_DEPTH_TEST );
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // ... OpenGL rendering ...

    // Ask rendition of the GigaVoxels pipeline
    _GigaVoxels->draw();
}
```

(c) Utilisation de GigaVoxels au sein d'une application OpenGL.

FIGURE 15 – « *Hello World!* » : déclaration (a), instanciation (b) et utilisation (c) d'un pipeline permettant de facilement intégrer GigaVoxels à une scène OpenGL.

6 Interopérabilité

6.1 Présentation sommaire

Une autre limitation de GigaVoxels était le manque de moyens d'interopérabilité. Jusque là, GigaVoxels ne permettait d'utiliser que des `cudaGraphicsResource`, un type de l'API CUDA permettant de manipuler des objets OpenGL ou Direct3D après les avoir enregistré manuellement dans CUDA.

On ne peut utiliser que des `cudaGraphicsResource` linéaires, ce qui nous limite à l'utilisation de *buffers* (pas de texture), et cette ressource devait impérativement être utilisée à la fois comme entrée et comme sortie de GigaVoxels (Fig. 16).

Ces deux limitations n'empêchaient pas l'utilisation de GigaVoxels qui était faite jusque-là, consistant à ne faire qu'un seul rendu dans un PBO (*Pixel Buffer Object*) OpenGL que l'on affiche ensuite à l'écran[13]. Ce manque d'API est surtout très contraignant pour un utilisateur voulant intégrer GigaVoxels à un pipeline complexe, mais on verra également, par la suite, qu'il est possible d'obtenir de meilleures performances en utilisant d'autres objets OpenGL que des PBO.

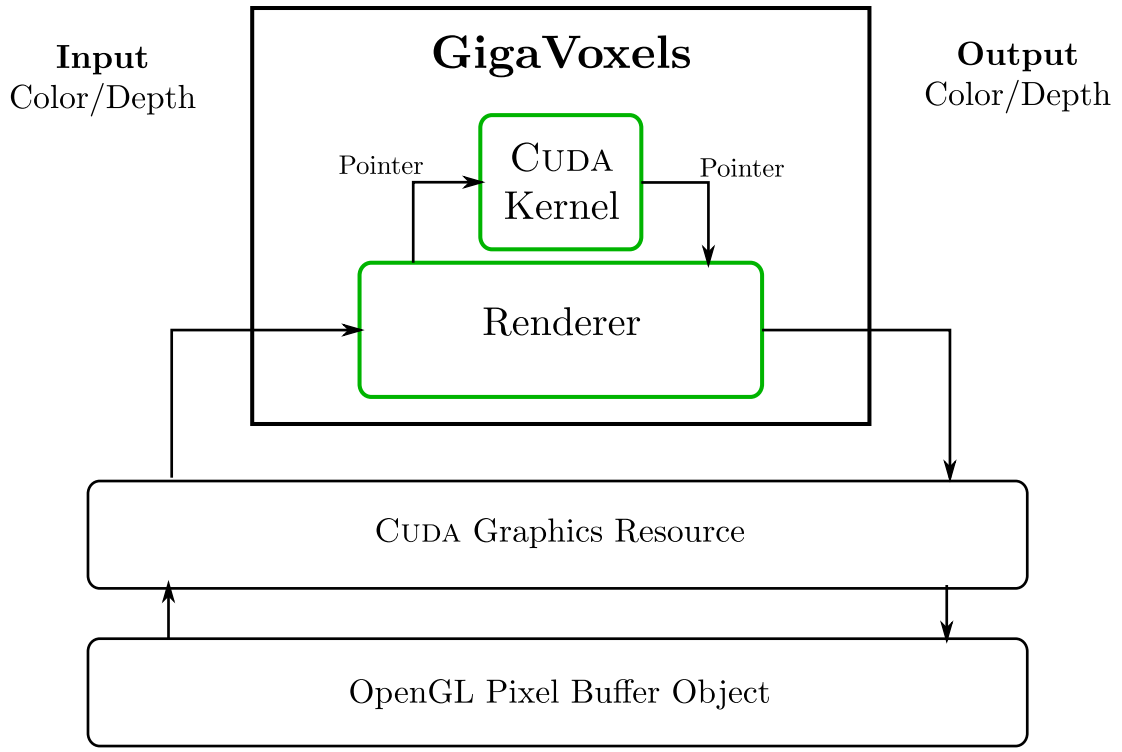


FIGURE 16 – Diagramme de l'ancien *renderer* ne pouvant utiliser que des CUDA *Graphics Resources* servant à la fois en entrée et en sortie.

13. Il est impossible d'accéder directement à l'écran.

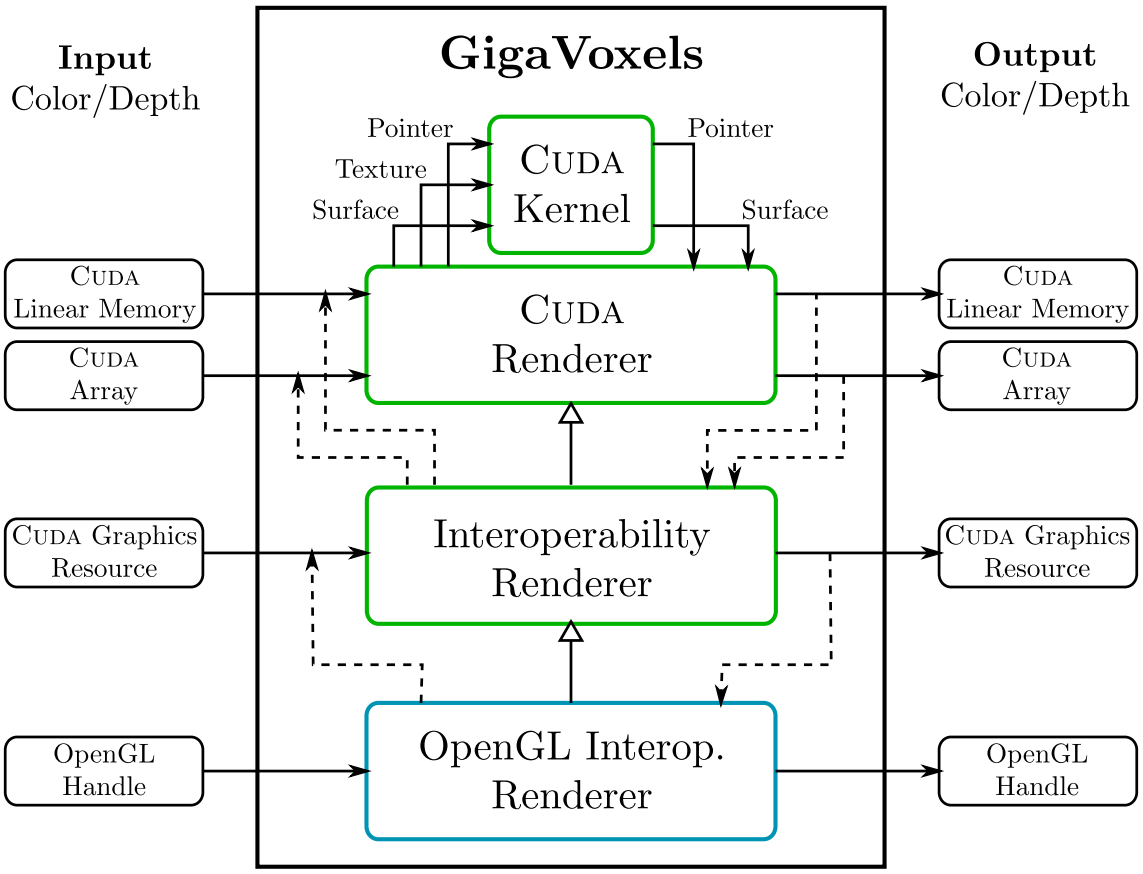


FIGURE 17 – Diagramme d'un nouveau renderer offrant la possibilité d'utiliser directement de la mémoire CUDA et, pour l'interopérabilité avec OpenGL, des PBO mais aussi des textures et des *render buffers*.

Hiérarchie de *renderers*

Il a été choisi de mettre en œuvre de nouvelles possibilités par la création de différentes classes *renderer* offrant de nouveaux types de stockage pour les entrées et sorties de GigaVoxels (Fig. 17). Faire de cette manière plutôt que de créer une classe dédiée a permis de faciliter le développement, mais on pourra par la suite mutualiser le code développé comme nous le verrons en fin de chapitre.

L'interface protégée est contrainte : une classe fille doit, en plus de la méthode publique permettant d'enregister un stockage, fournir trois méthodes protégées, dont certaines virtuelles. Seules ces dernières sont réellement indispensables au fonctionnement de l'objet. Toutefois, toutes aident une classe fille à fournir le support d'un nouveau type de ressource en s'appuyant sur l'existant.

L'ensemble sera présenté petit à petit, mais vous pouvez d'ores et déjà consulter le diagramme de classes ci-contre (Fig. 18).

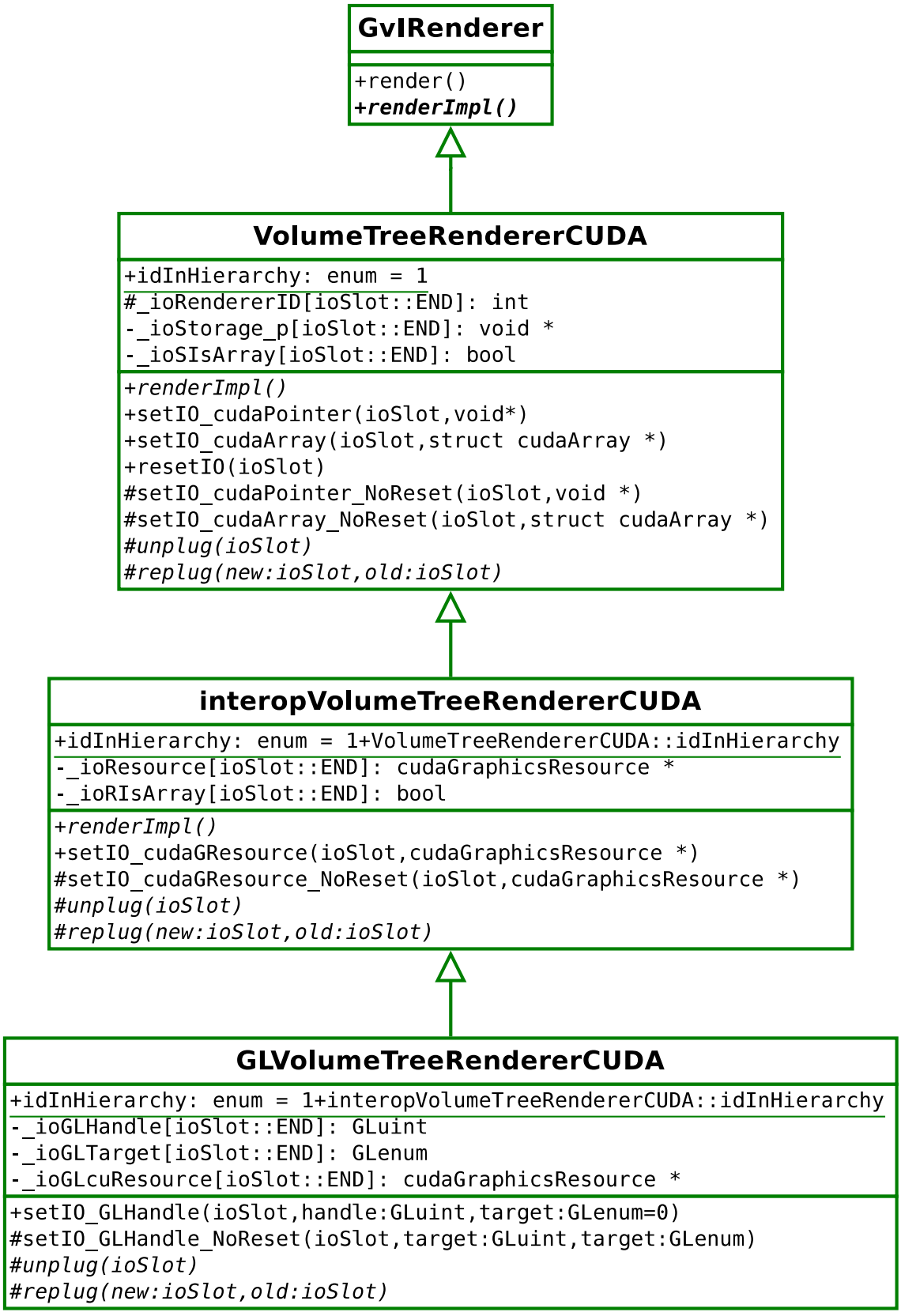


FIGURE 18 – Diagramme de classe des nouveaux *renderers* de GigaVoxels.

Différents canaux, différents *slots*

Chaque « canal » d'un renderer (RBGA, Z-buffer) dispose de trois « *slots* » : « entrée » ; « sortie » ; « entrée et sortie ». Pour des raisons de performance, le kernel de rendu ne doit pas avoir à gérer les différents cas pouvant survenir. Toutefois, seul le renderer de base s'occupe de l'interface avec le kernel et les collisions entre les différents *slots* est faite une fois pour toutes dans une seule méthode publique (`resetIO`) sur laquelle peuvent s'appuyer les classes filles.

Il est possible ainsi d'implémenter le support de nouveaux types de ressouces sans avoir à se soucier du risque d'effectuer en double les opérations préparant celles-ci, que ces opérations doivent n'être faites qu'une fois (OpenGL *handles*) ou avant chaque nouveau rendu (CUDA *graphics resources*).

On obtient au final une interface publique simple à utiliser (Fig. 19) permettant de plus de mélanger différents types de stockage. On peut par exemple utiliser des objets OpenGL en entrée pour effectuer le rendu dans un tableau CUDA, pour traitement ultérieur. Cette interface est par ailleurs extensible grâce à une interface protégée simple et claire évitant d'avoir à réimplémenter la gestion des collisions entre les différents *slots*.

Développements futur

Pour améliorer l'interopérabilité, il faudrait également, par exemple, pouvoir tout aussi simplement fournir ou récupérer la couleur en RBGA qu'en GBRA. Le système actuel ne le permet pas encore, mais c'est une amélioration qui ne serait pas si difficile à mettre en œuvre puisque la principale préoccupation est d'éviter d'avoir trop de branchements conditionnels au sein du *kernel* de rendu. Cela n'a pas été fait durant le stage car ce n'était pas une priorité.

6.2 Entrée et sortie

L'une des clefs de la solution choisie est la classe `GvRenderer::ioSlot`. Celle-ci facilite énormément la mise en œuvre et l'utilisation du *renderer* en permettant de faire abstraction du nombre de *slots* et de leurs types (couleur, profondeur, ...).

Un objet de type `ioSlot` peut prendre les valeurs `iColor`, `oColor`, `iDepth`, `oDepth`, `ioColor` et `ioDepth` représentant chacune une « prise » à laquelle il est possible de brancher un stockage. L'utilisateur peut ainsi facilement affecter ou réaffecter des ressources. Il peut, par exemple, enregistrer une ressource comme `ioColor` puis appeler `IOReset(ioSlot::iColor)` pour qu'elle ne soit plus utilisée qu'en sortie du *renderer*.

```
template<typename NodeRes,typename RealBrickRes,typename TDataList,
    typename VolumeTreeType, typename VolumeTreeCacheType,
    typename ProducerType, typename SampleShader,
    typename VolumeTreeRendererType>
void GvGLSimplePipeline< NodeRes, RealBrickRes, TDataList,
    VolumeTreeType, VolumeTreeCacheType, ProducerType, SampleShader, VolumeTreeRendererType >
::rebindIO()
{
    // reset input and output from GigaVoxels before modifying them
    this->_volumeTreeRenderer->resetIO( GvRenderer::ioSlot::iDepth );
    this->_volumeTreeRenderer->resetIO( GvRenderer::ioSlot::oColor );

    // set up input PBO
    glBindBuffer( GL_PIXEL_PACK_BUFFER, _idepth_pbo);
    glBufferData( GL_PIXEL_PACK_BUFFER, _width * _height * 4, NULL, GL_DYNAMIC_READ );
    glBindBuffer( GL_PIXEL_PACK_BUFFER, 0 );

    // set up output texture
    glBindTexture( GL_TEXTURE_RECTANGLE, _ocolor_tex );
    glTexImage2D( GL_TEXTURE_RECTANGLE, 0, GL_RGBA8, _width, _height, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL );
    glBindTexture( GL_TEXTURE_RECTANGLE, 0 );

    // set up the new input/output of the GigaVoxels renderer
    this->_volumeTreeRenderer->setIO_GLHandle( GvRenderer::ioSlot::iDepth, _idepth_pbo);
    this->_volumeTreeRenderer->setIO_GLHandle( GvRenderer::ioSlot::oColor, _ocolor_tex, GL_TEXTURE_RECTANGLE );
}
```

FIGURE 19 – Utilisation de la nouvelle API dans le pipeline présenté au chapitre précédant (Section 5.2, page 21).

Une instance de cette classe peut prendre également les valeurs `ioSlot::BEGIN`, `ioSlot::FIRST_IO` et `ioSlot::END`. Ces trois dernières valeurs permettent non seulement aux *renderers* d'allouer automatiquement des tableaux de la taille nécessaire mais aussi et surtout d'effectuer des boucles sur ceux-ci (Fig. 20), la classe `ioSlot` offrant une série de petites méthodes permettant de caractériser l'instance (`isOutput()`, `isOutputOnly()`, `isIO()`, ...), mais également d'en récupérer une autre lui étant associée (`getOutput()`, `getIO()`, ...) (Fig. 21).

Cela a permis d'implémenter les renderers de façon générique : indépendament du type du canal traité (couleur, prodondeur, ...), mais aussi du nombre total de canaux à traiter. Il est à présent possible de rajouter un nouveau canal d'entrée très facilement en ne modifiant que la classe `ioSlot` et, bien évidemment, le *kernel* de rendu qui accède aux ressources.

6.3 Hériter d'un *renderer*

La mise en œuvre de la méthode `VolumeTreeRendererCUDA::resetIO` (Fig. 21) nous amène à la conception de l'héritage des *renderers*. Celui-ci a été conçu pour que chaque nouvelle classe de *renderer* puisse offrir un nouveau type de ressource en s'appuyant sur ceux existants.

Pour cela, en plus de fournir une nouvelle méthode publique pour enregistrer une nouvelle resource en entrée/sortie, il faut veiller à définir trois méthodes protégées :

1. `virtual void unplugIO( ioSlot )`
2. `virtual void replugIO( ioSlot, ioSlot )`
3. `void setIO_<...>_NOReset( ioslot, ... )`

Remise à zéro d'un *slot*

Le débranchement d'une ressource se fait à l'aide de la méthode `resetIO` (Fig. 21) qui s'occupe seule d'assurer la cohérence des entrées/sorties du *renderer*. Par exemple, si une ressource est enregistrée comme `ioColor` et que l'on effectue l'appel `IOReset(ioSlot::iColor)`, alors la ressource est réaffectée en sortie seule du *renderer*. Par ailleurs, cet appel de `IOReset(ioSlot::ioColor)` permet de s'assurer tout autant de débrancher une ressource placée en entrée, qu'une ressource placée en sortie, même s'il s'agit de deux stockages différents.

Cette méthode non virtuelle est capable de remplir ce rôle parce qu'elle utilise les méthodes `unplugIO()` et `replugIO()` citées précédemment qui, elles, sont virtuelles. En héritant un *renderer*, on ne doit donc implémenter que ces deux fonctions, faciles à mettre en œuvre, pour que la méthode `resetIO()` de la classe de base puissse continuer à faire son travail.

```
template< typename VolumeTreeType, typename VolumeTreeCacheType,
    typename ProducerType, typename SampleShader >
GLVolumeTreeRendererCUDA< VolumeTreeType, VolumeTreeCacheType,
    ProducerType, SampleShader >
::~GLVolumeTreeRendererCUDA()
{
    for(ioSlot slot = ioSlot::BEGIN ; slot != ioSlot::END ; ++slot)
    {
        if ( this->_ioRendererID[ slot ] != 0 )
            this->IOReset( slot );
    }
}
```

FIGURE 20 – Utilisation d'une boucle sur les différentes instances possibles de la classe `ioSlot` pour la mise en œuvre du destructeur de la classe `GLVolumeTreeRendererCUDA`.

```
template< typename VolumeTreeType, typename VolumeTreeCacheType,
    typename ProducerType, typename SampleShader >
void VolumeTreeRendererCUDA< VolumeTreeType, VolumeTreeCacheType,
    ProducerType, SampleShader >
::resetIO( ioSlot slot )
{
    unplugIO( slot );
    if ( slot.isInputOnly() )
    {
        // if IO is set, set it to output
        replugIO( slot.getIO(), slot.getOutput() );
    }
    else if ( slot.isOutputOnly() )
    {
        // if IO is set, set it to input
        replugIO( slot.getIO(), slot.getInput() );
    }
    else
    {
        // reset input and output
        unplugIO( slot.getInput() );
        unplugIO( slot.getOutput() );
    }
}
```

FIGURE 21 – Utilisation de la classe `ioSlot` pour la mise en œuvre de la méthode `VolumeTreeRendererCUDA::resetIO`. Cette méthode non virtuelle repose sur l'utilisation des méthodes virtuelles `unplugIO` et `replugIO`.

```
template< typename VolumeTreeType, typename VolumeTreeCacheType,
    typename ProducerType, typename SampleShader >
void VolumeTreeRendererCUDA< VolumeTreeType, VolumeTreeCacheType,
    ProducerType, SampleShader >
::setIO_cudaPointer( ioSlot slot, void *p )
{
    resetIO( slot );
    setIO_cudaPointer_NoReset( slot, p );
    _ioRendererID[ slot ] = idInHierarchy;
}
```

FIGURE 22 – Mise en œuvre de `setIO_cudaPointer` reposant sur `resetIO` et la méthode protégée `setIO_cudaPointer_NoReset`. Cette dernière est elle-même utilisée dans la classe fille pour la mise en œuvre de la méthode `setIO_cudaGResource`

Supporter un nouveau type de ressource

La troisème méthode protégée, enfin, permet d'enregistrer une ressource dans le *renderer* sans se préoccuper de la hiérarchie à laquelle cette classe appartient, c'est-à-dire sans interagir avec aucune d'entre elles. Cette méthode ne fait donc pas de *reset* et ne change aucun état dans les autres classes de cette hiérarchie.

Cette méthode est d'ores et déjà extrêment utile pour la mise en œuvre des autres méthodes de la classe, publiques (Fig. 22) ou privées. Elle permet également à une classe fille de fournir le support d'un nouveau type de ressource en s'appuyant sur sa classe mère.

Ainsi, pour utiliser des objets OpenGL, la classe `GLVolumeTreeRendererCUDA` profite de la méthode `setIO_cudaGResource_NoReset()` fournie par la classe dont elle hérite, `interopVolumeTreeRendererCUDA`. Ensuite, cette dernière s'appuie à son tour sur les méthodes de sa classe mère, `setIO_cudaPointer_NoReset()` et `setIO_cudaArray_NoReset()`.

6.4 Transmission des données au *kernel* de rendu

On touche ici à un point critique de l'interopérabilité en termes de performance. Il faut toutefois composer avec certaines limitations de l'API de CUDA permettant de manipuler la *texture memory*.

Cette dernière doit obligatoirement être accédée au travers d'une *surface reference* ou d'une *texture reference*. Elles doivent impérativement être définies dans l'espace global et ont implicitement une qualification statique limitant leurs portées à l'unité de compilation.

Lors ces accès en mémoire, que ce soit en lecture ou en écriture, la référence utilisée doit impérativement avec CUDA 4.2 être un identificateur simple, connu à la compilation. Il est du coup impossible de passer ces objets en paramètre d'une fonction par un pointeur. On ne peut utiliser une référence déterminée dynamiquement qu'à partir d'une chaîne de caractère correspondant à son identificateur, et cela est trop coûteux pour pouvoir être fait dans le *kernel* de rendu.

Ensuite, le nombre de ces *surface references* que l'on peut utiliser en même temps est particulièrement limité [14], ce qui nous oblige à utiliser des *texture references*. Ces dernières doivent toutefois être déclarées différemment selon leurs dimensions et le type des données à accéder, ce qui oblige à en manipuler plusieurs par *slot*.

Ces limitations sont contraignantes, mais pour pouvoir exploiter les textures et RBO d'OpenGL et, ainsi, obtenir les meilleures performances, on est obligé d'utiliser la *texture memory*. S'il n'est pas possible de développer un système de stockage complètement générique, on peut tout de même développer un système spécialisé comme l'est déjà le cache interne à GigaVoxels.

Il faut de plus veiller à déclarer toutes les *texture/surface references* puis a définir des *macros* permettant de retrouver l'identificateur nécessaire à partir d'arguments. Ceux-ci devront toutefois être connus à la compilation pour que cette dernière puisse aboutir.

6.5 Stockage *template*

On a à présent une API agréable à utiliser comme on l'a vu précédement (Fig. 19) mais également facile a étendre que l'on veuille ajouter le support d'un nouveau type de ressource ou de nouveaux canaux. Elle est de plus simple et concise (Fig. 18) mais, maintenant que l'on a un système pleinement opérationnel, il serait intéressant de modifier l'architecture pour séparer le *renderer* de l'API qui vient d'être conçue.

Pour rester dans l'esprit dans lequel a été fait GigaVoxels jusque-là, on peut développer un type de stockage externe au renderer auquel il sera transmis en tant que paramètre `template` de sa classe de base `gvIRenderer`. Cela permettra, lorsque l'on aura un second *renderer* fonctionnel, de mutualiser le code de l'interface avec celui-ci plus facilement. Cela permet d'ailleurs de bien cloisonner le code relatif aux entrées/sorties du reste du *renderer*.

N'étant pas une priorité, cela n'a pas été mis en œuvre, mais le diagramme UML suivant (Fig. 23) décrit précisément une démarche envisagée.

14. Seules 128 *texture references* et 8 *surface references* peuvent être accéssibles en même temps avec les GPU de *compute capability 2.0* ([4], *annexe F.1*), sur lesquels doit fonctionner GigaVoxels.

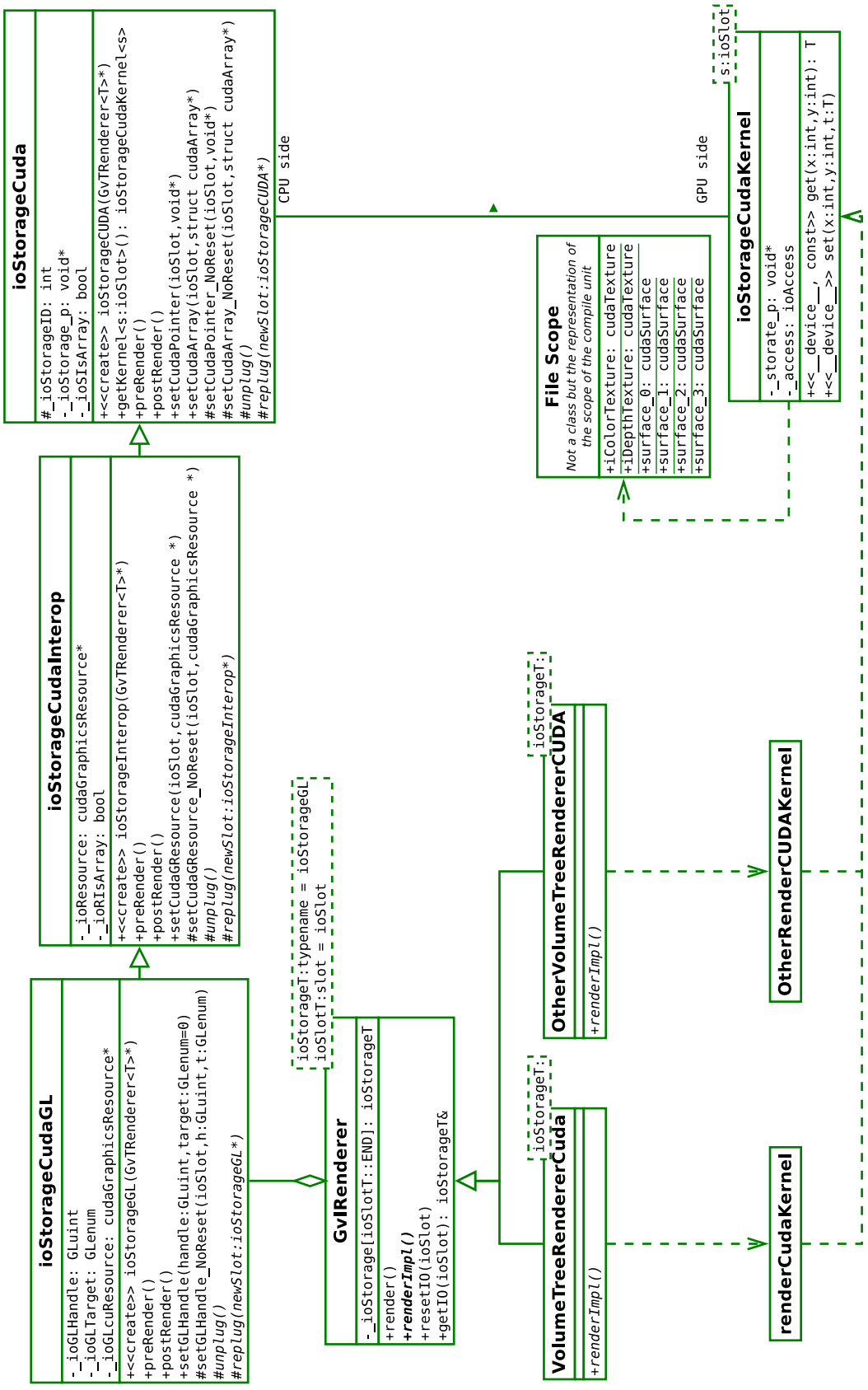


FIGURE 23 – Diagramme de classe du stockage à passer en paramètre *template*.

6.6 Performances de l'interopérablité

L'API développée permet à l'utilisateur final de choisir quelles ressources utiliser, mais il est important de pouvoir mesurer l'impact de ces choix sur les performances.

Il a donc été développé un programme permettant de changer à la volée la méthode utilisée (Fig. 24).

Cela a notamment permis de déterminer quelle méthode offrait les meilleures performances, mais aussi laquelle offrait le meilleur compromis entre performances et facilité d'utilisation (Fig. 25).

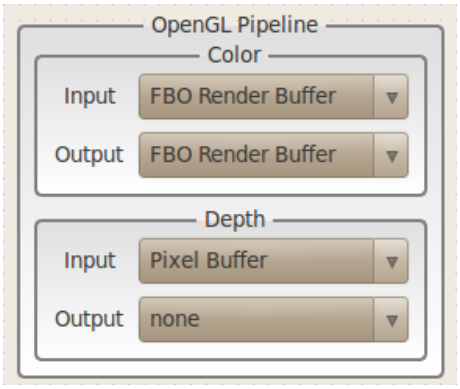


FIGURE 24 – Choix des objets utilisés pour l'interopérabilité.

Les performances des méthodes retenues ont été écrites en gras. Celles-ci, numérotées de 1 à 3 sont décrites ci-après. Chacune de ces méthodes offre un compromis différent entre sa facilité d'utilisation, ses performances et la possibilité de mêler GigaVoxels avec une scène en triangles.

input color	output color			
	PBO	Tex	RB	input
PBO	180	181	181	180
Tex	186	187	187	187
RB	185	187	187	187
∅	187	189	189	190

(a) Deux PBO de profondeur en entrée puis en sortie.

input color	output color			
	PBO	Tex	RB	input
PBO	184	186	185	185
Tex	189	190	190	190
RB	189	190	190	190
∅	192	193	193	196

(b) Un PBO de profondeur en sortie.

input color	output color			
	PBO	Tex	RB	input
PBO	242	242	242	242
Tex	242	253	253	253
RB	242	253	**254**[2]	253
∅	242	**251**[3]	253	258

(c) Un PBO de profondeur en entrée.

input color	output color			
	PBO	Tex	RB	input
PBO	256	254	256	256
Tex	261	260	260	260
RB	261	260	261	260
∅	268	**270**[1]	267	276

(d) Aucune entrée/sortie de profondeur.

FIGURE 25 – Tableau répertoriant les performances (images par seconde) de l'interopérabilité en fonction des méthodes d'entrée/sorties avec une fenêtre de 512x512 pixels, sans rendu. Sont en gras les méthodes les plus intéressantes.

1. Utilisation d'une texture en sortie pour la couleur, sans utiliser la profondeur.

   Cette méthode offre les meilleures performances dans le cas où l'on souhaite afficher le rendu de GigaVoxels tel quel, sans l'intégrer à une scène en triangles.

2. Utilisation de *render buffers* en entrée/sortie pour la couleur et d'un PBO en entrée pour la profondeur.

   Cette méthode offre les meilleures performances dans le cas où l'on souhaite intégrer GigaVoxels à une scène en triangles, tout en offrant le meilleur résultat en cas d'intersection entre les deux, GigaVoxels s'occupant de faire l'*anti-aliasing*.

3. Utilisation d'un PBO en entrée pour la profondeur puis d'une texture en sortie pour la couleur [15].

   Cette méthode offre les meilleures performances dans le cas où l'on souhaite intégrer GigaVoxels à une scène en triangles, tout en offrant le meilleur résultat en cas d'intersection entre les deux, GigaVoxels s'occupant de faire l'*anti-aliasing*.

15. La texture générée est semi-transparente et OpenGl s'occupe de faire l'*alpha-blending*.

7 Conclusion

On a au final résolu les problèmes que l'on s'était posés initialement puisque GigaVoxels permet à présent de mettre en place des scènes hétérogènes mêlant voxels et triangles. On dispose également d'un prototype de *geometry instancing* pleinement fonctionnel et l'on peut facilement générer des hyper-textures.

L'utilisation du moteur est également grandement facilité par la mise en place de deux API claires et bien documentées. La première, de haut niveau, permet d'utiliser le moteur très simplement avec un « Hello world ! » nécessitant l'ajout d'une quinzaine de lignes de code seulement. La deuxième API, de bas niveau, permet d'intégrer GigaVoxels à un pipeline graphique complèxe grâce à la possibilité d'utiliser différents types de ressource et, notamment, de nombreux objets OpenGL.

8 Annexe

A Présentation de Cuda

Cuda (*Compute Unified Device Architecture*) est une architecture matérielle de GPUs (*Graphical Processor Unit*) développée par Nvidia [16] qui l'utilise maintenant pour tous ses GPU. Cette architecture facilite la programmation de ces GPU par l'intermédiaire d'une API [17] dédiée, de deux languages, « Cuda C » et « Cuda Fortran », et de leur compilateur Nvcc.

Ces deux languages offrent des sur-ensembles des languages C et Fortran conçus pour être immédiatement utilisables au sein d'une application elle-même écrite en C, en C++ ou en Fortran. Il faut tout de même noter que Cuda C offre la plupart des fonctionnalités du C++, il y a tout de même des exceptions notables [18].

Par abus de language, « Cuda » est également utilisé pour désigner les languages de programmation sus-cités et les API associées.

Cuda kernels

Le cœur d'une application écrite avec Cuda est le *kernel*, le sous-programme exécuté à proprement parlé sur le GPU. Celui-ci est exécuté par une grille de blocs, chaque bloc pouvant lui-même être composé de quelques dizaines à quelques centaines de *threads*. Les dimensions de la grille et des blocs qui la composent, de 1D à 3D, peuvent être choisies à l'exécution du programme. Cela offre un cadre pouvant tirer parti de la puissance des GPU pour l'écriture de codes massivement parallèles.

Contraintes

Cuda permet de tirer parti de la puissance des GPU, mais impose tout de même certaines contraintes.

Une des premières avec laquelle se familiariser est la notion de *warp*. Un *warp* est un groupe de *threads* d'un même blocs se trouvant unis au sein d'un microprocesseur. Sa taille varie en fonction du GPU et il est indispensable d'éviter toute divergence au sein d'un warp, lors d'un branchement conditionnel par exemple. Dans le cas contraire, chaque thread d'un warp devant exécuter la même instruction, charque partie du *warp* devra attendre pendant que l'autre exécute son code.

16. `http://www.nvidia.fr/object/what_is_cuda_new_fr.html`
17. « *Application Programming Interface* » ou « Interface de programation ».
18. Annexe D.2 du « Cuda C Programming Guide »[4]

En sus, et à titre d'illustration, là ou l'on n'a pas à manipuler différent types de mémoire sur CPU, on se retrouve à devoir en manipuler de nombreux types différents en programmant avec CUDA :

1. *local memory* ;
2. *shared memory* ;
3. *global memory* ;
4. *texture memory* ;
5. *constant memory* ;
6. *pinned host memory* ;
7. *host memory*.

Chaque type de mémoire est disponible en quantités différentes, et offre des temps et des modalités d'accès très différents que ce soit en lecture ou en écriture[4][5].

Références

[1] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, Elmar Eisemann : Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press.
http://maverick.inria.fr/Publications/2009/CNLE09/

[2] Cyril Crassin : Gigavoxels : A voxel-based rendering pipeline for efficient exploration of large and detailed scenes. *Grenoble University Ph D. Thesis*, July 2011.
http://maverick.inria.fr/Membres/Cyril.Crassin/thesis/

[3] Reynald Arnerin, Fabrice Neyret : A journey in a procedural volume optimization and filtering of perlin noise, 2009.
http://maverick.inria.fr/Publications/2009/AN09

[4] Nvidia Corporation : *CUDA C Programing guide*.
http://www.nvidia.com/content/cuda/cuda-documentation.html

[5] Nvidia Corporation : *CUDA C Best Practices Guide*.
http://www.nvidia.com/content/cuda/cuda-documentation.html

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)